Noname manuscript No.
(will be inserted by the editor)

Flexible Partitioning for Selective Binary Theta-Joins in a Massively Parallel Setting

Ioannis Koumarelas · Athanasios Naskos · Anastasios Gounaris

Received: date / Accepted: date

Abstract Efficient join processing plays an important role in big data analysis. In this work, we focus on generic theta joins in a massively parallel environment, such as MapReduce and Spark. Theta joins are notoriously slow due to their inherent quadratic complexity, even when their selectivity is low, e.g., 1%. The main performance bottleneck differs between cases, and is due to any of the following factors or their combination: amount of data being shuffled, memory load on reducers, or computation load on reducers. We propose an ensemble-based partitioning approach that tackles all three aspects. In this way, we can save communication cost, we better respect the memory and computation limitations of reducers and overall, we reduce the total execution time. The key idea behind our partitioning is to cluster join key values following two techniques, namely matrix re-arrangement and agglomerative clustering. These techniques can run either in isolation or in combination. We present thorough experimental results using both band queries on real data and arbitrary synthetic predicates. We show that we can save up to 45% of the communication cost and reduce the computation load of a single reducer up to 50% in band queries, whereas the savings are up to 74% and 80%, respectively, in queries with arbitrary theta predicates. Apart from being effective, the potential benefits of our approach can be estimated before execution from metadata, which allows for informed partitioning decisions. Finally, our solutions are flexible in that they can account for any weighted combination of the three bottleneck factors.

I. Koumarelas
Department of Informatics, Hasso-Plattner-Institut, Potsdam, Germany
E-mail: ioannis.koumarelas@hpi.de

A. Naskos and A. Gounaris
Department of Informatics, Aristotle university of Thessaloniki, Greece
E-mail: {anaskos,gounaria}@csd.auth.gr

1 Introduction

Traditionally, analytical queries have played a significant role in a wide spectrum of data analysis techniques, spanning from simple statistics generation to data warehousing and support for data mining algorithms. For very large data volumes, like those derived from scientific experiments or extracted from web-server logs, one of the most common approaches is to perform the analysis according to the MapReduce programming model, and its descendants, such as Spark. Not surprisingly, analytical query processing in MapReduce has attracted a lot of interest, and the relevant work has investigated several issues, including indexing, data placement, data layouts, optimizations, iterative processing, fair load allocation and interactive processing to name some of them [11,19]. In this work, we focus on improving the efficiency of join queries executed in a massively parallel setting, and more specifically, we target selective binary theta-joins. Selectivity is defined as the ratio of the join result records to the results of the cartesian product; selective binary joins are those with low selectivity values. Theta-joins generalize equi-joins in the sense that the join condition between two datasets is arbitrarily complex rather than a simple equality constraint.

Join processing in a MapReduce environment has been extensively investigated recently [21,3,30]. The reference algorithm for partitioning the work of binary theta-joins to MapReduce reducers, presented in [21], is accompanied by optimality theoretical guarantees [16]. The fact that the solution in [21] is optimal according to the analysis in [16] may be deemed as discouraging for conducting further research on this topic. But in this work we show that, when we consider *selective* theta-joins of the form $S \bowtie_\theta T$, there is a significant space for improvement in terms of (i) the communication cost between the map and the reduce phase and (ii) the maximum memory and computational load a single reducer receives. The memory load is measured in terms of the input size of each reducer, and the computational load is measured in terms of the tuple pairs for which the theta predicate is checked at runtime (either implicitly or explicitly depending on the actual join algorithm employed at each reducer locally). These aspects are directly related to the total cost and the running time of a MapReduce/Spark application for theta joins. In this work we manage to significantly enhance the reference algorithm in [21] for joins with relatively low selectivity, e.g., up to 30%.

The main challenge in efficient processing of a parallel theta join stems from the inherent trade-off between the (maximum) number of records a reducer becomes responsible for and the replication factor between the map and the reduce phase. The replication of the map output across several reducers is necessary to ensure result correctness [21,22,2] and determines the communication cost between mappers and reducers. The replication issue is mitigated if fewer reducers are used. In the extreme case, where there is a single reducer, there is no replication; however, since a single reducer becomes responsible for the whole dataset, it may run out of memory both during processing its input and producing its output.

Current solutions, including the reference algorithm mentioned above, target a single criterion at a time, e.g., maximum reducer input [21,12] with the exception of [27], which considers both input size and computation load per reducer. Our first main novelty is that we consider all three important factors, namely the replication rate, the maximum reducer input and the maximum computation load per reducer in a configurable way. Furthermore, we significantly improve efficiency even when a single criterion is taken into account. Broadly, the problem we examine is, given the number of reducers that are available, to derive a partitioning that yields improvements in both the replication factor and the maximum workload allocated to a reducer in terms of memory and computation load.

A key element of our approach is the manipulation of the *join matrix (JM)*; the *JM* is a binary two-dimensional array that captures the information about which set of values from S match those from T according to the θ condition. It is straightforward to construct such a *JM*, where the values of S and T are ordered, and then, to pass it as an input to the algorithms in [21] to partition it to sub-matrices, one for each reducer (see Section 2 for more details).

Our approach is to cast the problem of partitioning parallel theta-joins as the task of partitioning the *JM* according to multiple criteria. In order to partition the *JM*, we follow two methodologies. The first one uses an existing partitioner but pre-processes the *JM*, so that its contents are clustered with a view to facilitating the partitioner to take better decisions. The second methodology, contrary to existing approaches [21,12,27], departs from assigning consecutive parts of the *JM* to a single partition, and performs agglomerative clustering on the *JM* contents based on a series of policies. Apart from being effective in improving on all bottleneck factors, some additional strong characteristics of our proposal are as follows:

- It is flexible in that it can consider any weighted combination of replication, and memory and computation load on reducers.
- The impact of our techniques can be accurately estimated before real execution. This means that the proposed partitioning need not be enforced if estimated to be non-satisfactory.
- In principle, the two main methodologies can be combined, i.e., they need not be regarded as antagonistic to each other.
- Our techniques are platform-independent and can apply to frameworks, such as Hadoop, Spark, Flink and any other framework that implements or extends the MapReduce paradigm. However, here we provide example running times only for Spark.

In summary, the contribution of our work is three-fold:[1]

1. We investigate ensemble matrix re-arrangement techniques that perform clustering of *JM* values. The rationale behind the re-arrangement is that clustered *JMs* are amenable to more efficient workload distribution among reducers (see Section 4).

[1] In our 4-page abstract [16], we provide a preliminary version of the material in Section 4. All the remainder material in this work is novel.

2. We also propose a new ensemble methodology to partition the work among reducers that employs hierarchical agglomerative clustering. This methodology is capable of performing arbitrary partitioning and thus departs from the straightforward approach that allocates adjacent *JM* cells to a single reducer (see Section 5).
3. We thoroughly evaluate our techniques using joins with two types of theta predicates, namely predicates for band queries on real data and random ones. For band queries, the results show that we can save up to 45% communication cost, and reduce the maximum load a reducer receives by 50 %. These improvements are further increased in random theta-joins queries (74% and 80%, respectively). Finally, the running time of our approach is low, in the order of a few seconds, when executed on a single machine (see Section 6).

The remainder of this article is structured as follows. In Section 2, we provide background material on theta join processing in a massively parallel setting. The high level description of our approach and evidence of the impact of each of the bottleneck factors on Spark application running time are presented in Section 3. The details of the afore-mentioned contribution are in the following three sections. We discuss related work in Section 7 and we conclude in Section 8.

2 Background

We briefly describe the approach in [21] for evaluating binary theta joins $S \bowtie_\theta T$ in MapReduce, which is also applicable to Spark and other similar frameworks. The key underlying data structure is the *join matrix (JM)*. The *JM*, in its simple form, has as many rows as the cardinality of S and as many columns as the cardinality of T. Each cell of the *JM* thus corresponds to a pair of records from both datasets. The *JM* can take a more concise form where each row (resp. column) represents an equi-depth bucket and can thus refer to a range of values of the joining predicate of S (resp. T); this allows for a fixed *JM* size regardless of the cardinalities of the initial datasets.

Assuming a MapReduce setting, a parallel theta-join is executed as follows:
Map phase. For each tuple, assign the reducer it will be sent to as the first part of the complex map output key. The second part will refer to the candidate cells that will be explained shortly.
Shuffling phase. Assign keys to reducers based on their first part only.
Reduce phase. Execute local theta-joins.

Essentially, in the map phase, the *JM* is split into several regions, where each region is mapped to a reducer[2]. For each region, we can compute the amount of tuples that belong to it, which is the *input cost* of that region and is

[2] In the remainder of this work we will use the terms region, partition and group interchangeably; we will also use the term reducer for the worker node, where local join processing takes place, but this does not imply that we are tailored to a MapReduce setting only.

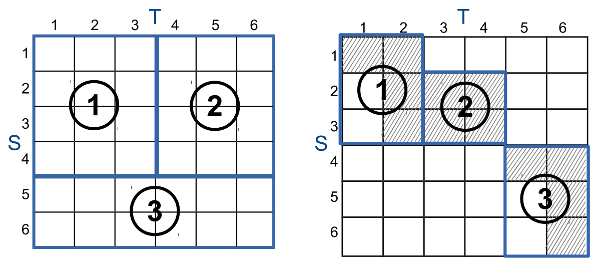


Fig. 1: Partitioning the JM in 1-Bucket-Theta (left) and M-Bucket (right).

directly related to the computation and memory load of the associated reducer. For perfect load balancing, we want these regions to have equal input cost and computation load. In order to accomplish the latter objective, two main algorithms are presented: 1-Bucket-Theta and M-Bucket-I (and its variation M-Bucket-O).

2.1 1-Bucket-Theta

1-Bucket-Theta is a generic algorithm that examines, either implicitly or explicitly, all tuple pairs and requires minimal statistical information, namely just the cardinalities of the input. A randomized technique is used during the partitioning process, according to which each value of S and T is randomly mapped to rows and columns, respectively. The JM is divided beforehand into regions of same size, based on the number of reducers available. The strong point of the algorithm is the principled way that it partitions the JM, in a way that all JM cells are covered and, at the same time, the maximum reducer input is minimized. The randomization alleviates the need of running an extra MapReduce pre-processing step to assign unique row and column numbers on each S and T tuple. Although this technique does not formally guarantee the desired input and output size for each reducer, there are insignificant statistical variations for large data sizes.

Figure 1(left) shows an exemplary partitioning across 3 reducers, where there are 6 tuples from S and 6 tuples from T, and the input cost of each reducer is 7 (4 tuples from S and 3 from T), 7 (4 from S and 3 from T) and 8 (2 from S and 6 from T), respectively. Overall, the reducers receive 22 tuples, whereas the total size of input is 12. The ratio of these two metrics $\frac{22}{12}$ denotes the *replication rate* for this specific partitioning and reveals the trade-off between lowering the input cost of each reducer through the increase in the number of reducers and increasing the number of reducers an input tuple has to be copied to. In the example, each tuple from T has to be copied twice, once for either the 1st or the 2nd reducer and once for the third one. Similarly, the first 4 tuples from S are copied twice to the first 2 reducers and only the 5th and 6th tuples from S are sent to a single reducer, the 3rd one. The computation load of a reducer denotes the number of pairs for which the θ condition needs to be checked and is equal to the number of JM cells. In the example, all reducers are responsible for 12 cells, i.e., there is perfect balancing regarding this criterion.

The algorithm is shown to be perfect for all-pairs comparisons and also suitable for high join selectivities. For lower selectivities, where a big portion of *JM* cells do not correspond to valid results (i.e., results that satisfy the theta join predicate), significant improvements can be attained capitalizing on the observation that some *JM* cells need not be assigned to any reducer. M-Bucket-I, discussed below, follows such an approach.

2.2 M-Bucket-I

The M-Bucket-I algorithm, which uses more statistical information, outperforms 1-Bucket-Theta in cases of low selectivity. M-Bucket-I employs a preprocessing step of histogram computation. This step runs two consecutive MapReduce jobs. The first job samples the values of the two sets and computes approximate k-quantiles, which are then used as the bucket boundaries of the equi-depth histogram, and the second job simply counts the actual number of tuples in each bucket. The *JM* denotes pairs of histogram buckets rather than pairs of tuples. Moreover, each pair of buckets is checked as to whether it can contain tuples that satisfy the join condition; the check is based on the bucket boundaries and the resulting cells are termed as *candidate* cells.

An example is shown in Figure 1(right). As previously, the *JM* is partitioned into rectangular regions which correspond to reducers input, but now the regions need not cover all the *JM* cells but only the candidate ones (shaded in the figure). In the example, if we assume that the depth of histograms are equal for both relations, the input of the three reducers is 5,4 and 5 buckets, respectively. Also, the replication rate is reduced to $\frac{14}{12}$. The computation load is 4 for all reducers.

The exact partitioning algorithm employs a binary search method to find the smallest upper bound on the reducer input for which a region splitting sub-routine can find a valid solution given the number of regions (i.e., reducers). That sub-routine splits the *JM* into several consecutive blocks of rows, and each such horizontal fragment is further split vertically. Each horizontal fragment is given a score which is equal to the average number of candidate cells in the vertical strides within the fragment. The subroutine detects the horizontal fragment with the maximum score starting from the top of the *JM*, and continues until either all rows are covered or the number of regions exceeds the number of reducers; the latter denotes an infeasible solution for the specific maximum reducer input bound.

The authors of [21] also propose a variation of M-Bucket-I algorithm called M-Bucket-O. The difference is that the former targets the minimization of the maximum reducer input, whereas the latter targets the minimization of the maximum reducer output. Note that estimating the reducer output based on histograms is prone to significant errors, even when the histograms are accurate. However, it holds that output is $O(number\ of\ candidate\ cells)$, and as such, the number of candidate cells can be used as an approximate metric for the output. In this work, we adopt a slightly different usage of $O(\textit{number of candidate cells})$, namely to denote the reducer computational load in terms of the number of tuple pairs that need to be checked, either explicitly or implicitly, during the actual query execution at the reducers. The reducers may employ any local join algorithm; the selection and/or development of the most efficient join algorithm for the local θ-predicate evaluation is orthogonal to our research.

3 Our approach

Consider a *JM* corresponding to the join $S \bowtie_\theta T$, the cells of which are grouped in P partitions. Then, for each partition $p \in P$, we define the input cost *IC(p)* and the number of the candidate cells *CC(p)*. *IC(p)* is equal to the number of the input tuples. The quality of the partitioning of the *JM* is assessed with the help of the following three metrics:

1. *replication rate (rep)*, formally defined as the ratio between the aggregate reducer input and the mapper input [2]: $rep = \frac{\sum_{p=1}^{P} IC(p)}{|S|+|T|}$;
2. *maximum reducer input (mri)*, defined as the maximum number of records sent to a single reducer: $mri = max(IC(p))$. *Mri* denotes the maximum memory load imposed to a single reducer (given that, for performance reasons, it is important each reducer to be capable of holding all its input in main memory);
3. *maximum reducer computation load (mrcl)*, defined as the maximum amount of pairs a single reducer checks (either implicitly or explicitly) to produce join results, which is proportional to the number of candidate cells in the *JM* (given that cells correspond to a pair of buckets from two equi-depth histograms): $mrcl = |Bucket(S)||Bucket(R)|max(CC(p))$. $|Bucket(S)|$ (resp. $|Bucket(R)|$) is the size in records of each equi-depth bucket of the histogram of S (resp. R).

The above metrics represent important aspects of data processing. In particular *rep* represents the communication (I/O) cost, *mri* corresponds to maximum required RAM so that processing takes place in main memory, and *mrcl* intends to represent the CPU load.

It is straightforward to measure sizes in bytes rather than number of records; also to define complementary metrics, by applying some other statistical function over the building blocks of the JM. In this work, we also investigate the *input imbalance (imb)*, defined as the ratio of *mri* to the average reducer input, considering only the non-idle reducers. Essentially, *imb* captures the deviation in input between the reducers[3]. To show that our three main metrics, namely *rep*, *mri* and *mrcl*, have a higher impact on performance

[3] It is also trivial to express *imb* as a function of *mri* and *rep* through simple algebraic manipulation.

than *imb*, consider the case in Figure 1 where we have 3 reducers, initially receiving 7, 7 and 8 records, respectively. If we manage to drop the input records to 5, 3 and 4, then, as shown in Figure 1(right), both *mri* and *rep* will significantly decrease, although *imb* will become 1.2 from 1.09. However, it is desirable to keep *imb* as close to 1 as possible. In the remainder of this work, we focus on the first three metrics, but we will also discuss imbalance.

The aim of our approach is to reduce an arbitrarily weighted combination of the replication rate and the maximum amount of memory and computational load a reducer receives, given the number of available reducers. Replication of the mappers' output to multiple reducers occurs whenever there are multiple candidate cells, either in the same row or the same column of the *JM* belonging to different regions (see for example the 2nd row in Figure 1(right)). Thus our rationale is to reduce the occurrences of such a situation without loading a single reducer too much following two types of techniques.

3.1 Theta joins on Spark: the impact of replication and reducer load

Before proceeding into the details of our approach, it is important to argue that aiming to optimize an arbitrarily weighted combination of *rep*, *mri* and *mrcl* is more general than aiming to minimize execution time using a set of queries and/or parallel infrastructures. More specifically, here we provide evidence that (i) *rep*, *mri* and *mrcl* have an impact on execution time, and (ii) the relative magnitude of their impact is application- and deployment-dependent, i.e., they depend on the data, theta predicates and infrastructure characteristics.

To this end, we conducted a series of experiments to show the impact of each of these metrics. We used the real-world dataset of the Cloud Dataset and the queries, as described in Section 6, with the cardinality of the two relations being 2 million records. For these queries, we created different partitioning plans according to the techniques in Sections 4 and 5. We randomly chose 32 pairs of plans corresponding to the same queries but differing in up to three out of the four metrics introduced in Section 3 (the three main metrics and the imbalance); this selection criterion helped us to verify that all metrics are important. Finally, we experimented with three flavors of raw datasets: *small*, where the raw tuples included only the join field, *medium*, where the raw tuples included also fields of size 1KB, and *large*, where the raw tuples were 5KB in size. The experiments ran on a Spark cluster at Barcelona Supercomputing Center, the details of which are described in [26], and employed 20 reducers. We allocated each reducer on a single core and in order not to cause network traffic unnecessarily, we made sure that these cores belong to two physical machines, each equipped with two 8-core Intel Xeon E5-2670 processors. The actual running times of the queries ranged from 49.9 to 940.6 secs, whereas the result sizes ranged from 4.1 to 27.6 billion records. Each query was executed 5 times and the median times are presented.

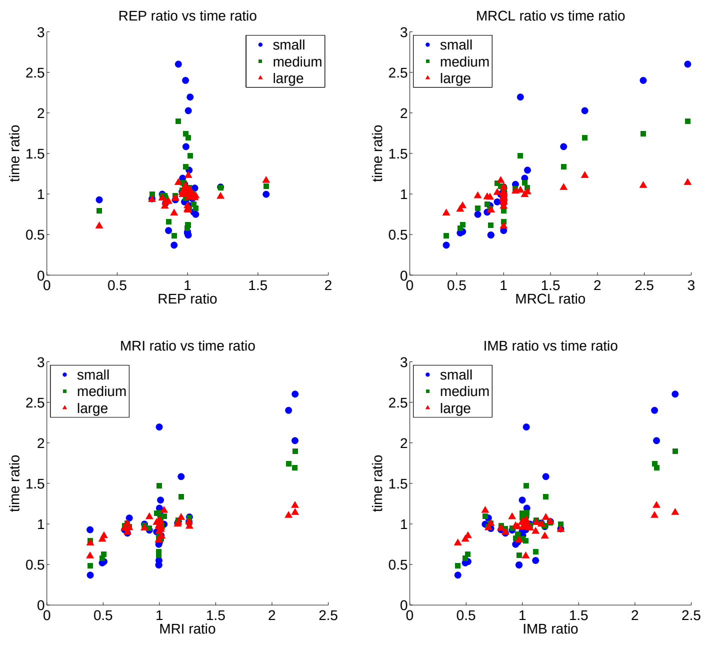


Fig. 2: Correlation of each of the four metrics and the response time.

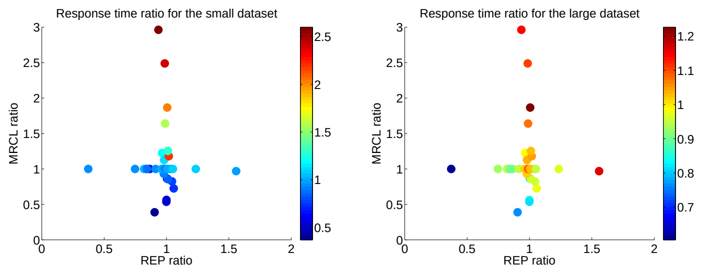


Fig. 3: Correlation between the response time and the combination of *rep* and *mrcl*.

Figure 2 shows some indicative aggregate results, where, for each of the 32 selected pairs, the correlation between the ratio of the response times and the ratio of the four metrics is examined. Starting from top-left, one can observe that there is a strong correlation between *rep* and the response time for the large dataset. This is expected since, the communication cost is more dominant when the tuples are larger in size. The correlation is significantly weaker for the other two datasets. Next, we can observe that *mrcl* is strongly correlated with the response time for the small and medium datasets. *Mri* and *imb* seem to be correlated with the response time as well. In addition, they exhibit the same behavioral pattern and thus we can safely consider only *mri*.

Figure 3 provides further insights into the relative impact of the *rep* and *mrcl* metrics on performance. Red colors correspond to higher ratios whereas blue colors correspond to lower ratios; the exact range is shown in the bars next to the images. On the left, we can see that, for the small dataset, the response time is mostly determined by *mrcl*, whereas for the large dataset both metrics are important. If we increase the number of reducers (no figures are shown), we have observed that the impact of *rep* is mitigated, that of *mri* slightly increases, and *mrcl* remains a dominant factor.

In summary, the main lesson from this set of experiments is that *rep, mri* and *mrcl* are indeed correlated to execution time and, for different queries and different data sizes, the weight of each of these three factors changes. Similarly, the cluster characteristics and settings, e.g., number of reducers, also affects the relative importance of the factors (no detailed results are presented). Also, these metrics are inherently related to the network transfer time, local disk I/O time, and CPU time, which are reported to be application and deployment-dependent in other recent works as well, e.g., [8]. Consequently, an objective function, in which the weights of the three metrics were fixed, would accurately reflect execution time only under certain conditions. On the contrary, aiming to optimize an arbitrarily weighted combination of the *rep, mri* and *mrcl* metrics renders our work more generally applicable, in the sense that it can cover execution times in arbitrary settings provided that the weights are appropriately set for such settings.

3.2 Problem Formulation and Approach Outline

Formally, the problem we target is specified as follows:

Problem Definition: Given (i) a maximum number of reducers P and (ii) a $n \times m$ *JM* matrix, find an allocation of each candidate cell $JM(i,j),\ 1 \leq i \leq n,\ 1 \leq\ j \leq m$ to a single reducer, s.t. the objective function

$$OF = \alpha \cdot rep + \beta \cdot mri + \gamma \cdot mrcl \tag{1}$$

is minimized, where α, β and γ are provided by the user. In Section 6, we show how the three metrics are normalized with regards to the baseline approach in Section 2.2 by appropriately setting the α, β and γ weights.

We explore two methodologies in order to allocate candidate cells to reducers more efficiently. First, we permute *JM*'s rows and columns in order to improve the quality of the partitioning phase; the latter can be performed according to M-Bucket-I/O. Intuitively, creating clusters of cells along the main diagonal will form a distribution of candidate cells inside the *JM*, which is expected to better fit in rectangular regions leading to no or little replication. Therefore, this technique essentially adds a step of beforehand analysis to the M-Bucket-I/O algorithm, just after the histograms are built and the initial *JM* is produced. We investigate several solutions for performing the re-arrangement.

Algorithm 1 High-level partitioning

```
Input: $JM(S \times T)$, $P$, $OF(\alpha, \beta, \gamma)$
  $bestValue, bestPartitioning \leftarrow NULL$
  for $r \in$ JM re-arrangement solutions do
    for $s \in$ JM candidate cell clustering solutions do
      $part \leftarrow$ partitioning based on $P$ and $r$ and $s$ solutions
      if $OF\ value\ of\ part < bestValue$ then
        $bestPartitioning \leftarrow part$
        $bestValue \leftarrow OF\ value\ of\ part$
  return $bestPartitioning$
```

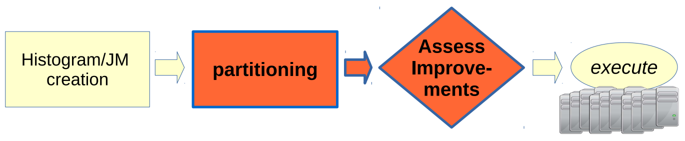


Fig. 4: The end-to-end theta-join processing. This work targets the two dark-shaded steps.

Second, we investigate a partitioning technique that is based on agglomerative clustering and is not restricted to rectangular areas or adjacent cells in the *JM*. This partitioning can be performed on either the original or the re-arranged *JM* before the actual execution. When partitioning the matrix, we consider all the factors that can potentially form a performance bottleneck. We investigate several policies for performing this.

A strong point of our approach is that the metrics mentioned above can be accurately estimated before the actual MapReduce/Spark program is executed by analyzing the final workload to be allocated to reducers. An implication of the fact that the quality of a partitioning can be accurately quantified before execution is that, whenever a new partitioning result is not estimated to yield any improvements, it can be simply discarded, and as such, our approach can be safely adopted. In the worst case, there will be an extra overhead for re-arranging the *JM* and derive non-beneficial partitioning, but this overhead is negligible, since it is polynomial in the *JM* matrix size. Figure 4 shows the phases of a theta-join execution, where our contribution refers to the shaded components. As shown, our techniques take place before the actual execution on a MapReduce or Spark platform.

The high level approach to partitioning that we follow is shown in Algorithm 1, which refers to the most generic case, where the two techniques investigated are used in combination. In the algorithm, *part* is a valid allocation of each *JM* to a reducer. The code of our work is available for download from `https://github.com/JohnKoumarelas/binarythetajoins` (assuming execution on Spark).

4 Re-arranging the Join Matrix

Our first type of techniques modifies the partitioning without actually proposing a new partitioning technique but through the proposal of using an existing partitioner after having modified the initial *JM*. The problem of cell re-arrangement can be addressed with several algorithm families, such as *clustering* (e.g., hierarchical, array-based, and so on), *combinatorial optimization* (e.g., bin packing, knapsack) and *bandwidth reduction*. In the present work we study the use of two widely known algorithms, the Bond Energy Algorithm [20] and the Traveling Salesman Problem (TSPk) algorithm. According to Algorithm 1, the presented solutions can form an ensemble and thus need not be regarded as antagonistic to each other.

4.1 BEA-based solutions

Bond energy algorithm (BEA) is a clustering technique used in a wide range of applications such as manufacturing (e.g., packing problem), distributed database design, and software engineering (e.g., for analysing program structure) [5].

The purpose of BEA is to identify and produce clusters of similar values in complex data arrays. This is accomplished by permuting rows and columns of an input data matrix in such a way so that the larger elements of the matrix are moved closer to each other. BEA uses a measure of effectiveness (ME) to validate the result of every permutation. The ME is computed as the sum of the bond strengths in the array, where the bond strength between two adjacent elements is defined as their product. More formally, $ME(A)$ is equal to:

$$\frac{1}{2}\sum_{i=1}^{i=N}\sum_{j=1}^{j=M} a_{ij}[a_{i,j+1}+a_{i,j-1}+a_{i+1,j}+a_{i-1,j}]$$

where A is any non negative $N \times M$ array, with the convention $(a_{0,j} = a_{N+1,j} = a_{i,0} = a_{i,M+1} = 0)$.

The BEA runs as follows. It initializes the new matrix by placing its first two rows (resp. columns). For each of the remaining rows $a_{i,*},\ 3 \leq i \leq N$ (resp. columns, $a_{*,i},\ 3 \leq i \leq M$), the algorithm examines its placement in the first i entries so that ME is minimized. When the algorithm finishes, the new array comprises the same rows (resp. columns) in a new order.

The BEA was selected among other clustering algorithms, due to its following characteristics that suit better to our problem specification: (i) BEA has polynomial complexity. (ii) It is applicable to matrices with non-equal dimensions. (iii) The row (resp. column) permutations are independent from the column (resp. row) permutation. Consequently, it only takes two passes to re-arrange the matrix. (iv) Finally, it tries to produce a block-diagonal matrix. Packing values closer to the diagonal of the JM reduces the replication factor (see Figure 1(right)).

$$\begin{vmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix} \begin{vmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{vmatrix}$$

(a) ME=2 (b) ME=4 (c) ME=8

Fig. 5: BEA Re-arrangement

The implementation of the BEA needs to be done carefully as it is prone to high execution times for large matrices. The algorithm makes two passes to re-arrange the matrix however, it also needs two iterations over every row (resp. column) item in every pass to compute the ME metric. To ameliorate this, we have compressed the JM using bitmaps instead.

In Figure 5, we present a simple example of the BEA applied on a 4×4 matrix. The ME values are shown below of every sub-figure. In Figure 5a, the original matrix given as an input to the algorithm is presented. The ME of the original matrix is 2. In Figure 5b, the resulting matrix after row permutations is presented where the ME is equal to 4. Last, in Figure 5c, the final matrix after the columns permutation is displayed with ME equal to 8.

We have also examined a variation of the BEA, where the values that are within a pre-specified radius are taken into consideration in order to compute the ME measurement, instead of only the upper (left) and bottom (right) ones. The results of this variation, termed as *BEARadius*, are presented in the experimental section.

4.2 TSP-based solutions

The authors of [9] recognize the extensive usability and superiority of BEA over other re-arrangement clustering algorithms, such as rank ordering clustering (ROC) [15] and direct clustering analysis (DCA) [5], but they point out a pitfall regarding the default ME measurement used in BEA. Giving concrete examples, they show that the ME used fails to distinguish between the levels of clustering of the pairs of zeros, producing inefficient clustering along the main diagonal in certain cases. They also present an example where this behavior is not limited to zeros. Their suggestion to overcome this pitfall and further issues that occur in other proposed algorithms, is to regard the TSP problem as a re-arrangement clustering one, given that these two problems are shown to be equivalent [18,17]. More specifically, [17] states that the BEA algorithm is "a simple suboptimal TSP method, which constructs a tour by successively inserting the cities", in the same way that BEA successively places rows (resp. columns).

More specifically, the TSP solution, which finds the circular tour with minimum distance among all complete tours, can be seen as a re-arrangement of the cities, in which most similar (i.e., close cities) are in adjacent places. This is equivalent to placing similar rows (resp. columns) in adjacent places, exactly as BEA aims to do with the convention that rows (resp. columns) represent

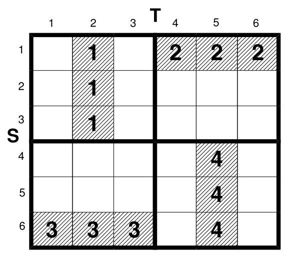


Fig. 6: A good example of splitting a JM in rectangular areas

cities. As an example, in Figure 5b, the last row of Figure 5a has been moved to the second place; this corresponds to a path, where the TSP solution visits the fourth city starting from the first one. To apply the TSP algorithm to the re-arrangement problem of our case, cycles need to be replaced with paths and the optimization problem of finding the minimum distance between the cities has to be replaced with the one previously presented in the BEA algorithm application, i.e. the maximization of ME. In [9], a mechanism to compute paths over cycles through the inclusion of dummy cities to the problem formation is presented. The replacement of the optimization problem is straightforward by replacing the respective objective function of the TSP algorithm with the ME(A) function presented earlier and inverting the minimization to a maximization problem. Any TSP solver can be used to solve the TSP problem; we preferred to use the Concorde TSP Solver[4] which is proved to be highly scalable, as it has solved TSP problems with up to 85900 cities. In the evaluation, we will refer to this solution as *TSPk*, where k is set to the number of available reducers.[5]

In the above solution, the re-arrangement algorithm permutes the rows, which correspond to the cities, and uses the columns only for the distance/dissimilarity computation. Given that in our problem formation, the rows and columns have equivalent role, we have created an additional re-arrangement algorithm, where we first provide the rows as cities to the TSP solver and then, we transpose the JM to provide the columns as cities. Thus the permutations are applied to both rows and columns. This variant is called TSPk-Transposed (*TSPkT*).

5 Partitioning the Join Matrix

Matrix re-arrangement aims to facilitate the existing partitioning technique in [21] to reduce the replication rate and the other metrics of interest; our partitioning solutions aim to tackle the same problem more directly. The role of the partitioner is to derive a mapping of candidate *JM* cells to reducers,

[4] `http://www.math.uwaterloo.ca/tsp/concorde/`

[5] *TSPk* is implemented according to [9], the code of which has been integrated into our codebase under the `https://github.com/JohnKoumarelas/binarythetajoins/tree/master/btj/tspk` directory.

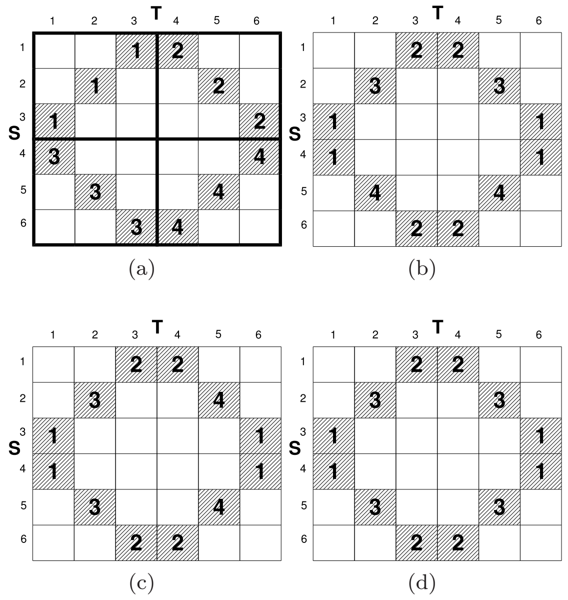


Fig. 7: A counter-example of splitting a JM in rectangular areas along with 3 more efficient partitioning solutions

which corresponds to a customized shuffling of map output. The mapping is implemented with the help of an associative array structure which maps every histogram bucket to a set of reducers. In Hadoop, it can be easily enforced through the `Job::setPartitionerClass` function. Alternatively, in any massively parallel framework that supports shuffling of key-value records based on key (e.g., through a `groubByKey` function), the partitioning can be enforced through setting the key to the reducer id.

5.1 Non-Rectangular Groups of Candidate Cells

The novelty of our approach is that it explores solutions that can group candidate cells to the same reducer even if those cells are far away in the *JM*. Consider the example in Figure 6, where grouping the cells in a rectangular way so that four regions are constructed (assuming the availability of four reducers) produces the best outcome in terms of the maximum input cost per reducer and the minimum replication rate. The reducer input cost equals four equi-depth histogram buckets, and on average, each input record is replicated $\frac{16}{12} = \frac{4}{3}$ times. The selectivity associated with that *JM* equals the percentage of candidate cells, which is $\frac{12}{36} = 0.33$. In the figure, the number in each candidate cell denotes the id of the reducer that cell is allocated to.

Consider now, in Figure 7a, another example with equal selectivity, where M-Bucket-I's partitioning is applied. We can easily compute that $mri = 6$,

Algorithm 2 Allocate JM candidate cells to reducers

```
Input: $JM(S \times T),\ P$
  $cand \leftarrow countCandidateCells(JM)$
  $mri\_min \leftarrow 2\sqrt{cand/P}$
  $mri\_max \leftarrow |S| + |T|$
  while $mri\_max > mri\_min$ do
    $mri \leftarrow \lceil \frac{mri\_max+mri\_min}{2} \rceil$
    if $nonRectangularSplitting(JM, P, mri)$ then
      $mri\_max \leftarrow mri$
    else
      $mri\_min \leftarrow mri + 1$
  return $mri$
```

and *rep* = 2. Figures 7b and 7c show two equivalent partitionings that cannot be displayed as rectangular regions but improve both on *rep* and *mri* at the expense of increasing the *mrcl*: *mri* becomes 4, the replication rate drops to $\frac{7}{6}$, but *mrcl* increases by one cell. Note that if we slightly modify our problem to allow for different number of reducers, we may come up with other efficient partitionings; for example, if we could consider 3 groups, as depicted in Figure 7d, *mri* drops to 4 input buckets as well, but there is no replication ($rep = 1$).

The examples above provide strong insights that non-rectangular grouping of candidate cells is a promising alternative. However, the problem of optimal partitioning is intractable. It generalizes the already NP-hard bin-packing problem. We do not only allocate candidate cells to reducers (which might be deemed as the classical bin-packing problem) but in addition allocations are correlated, since each allocation impacts on the quality of other allocations because of the replication rate.

Our solutions employ three optimization criteria, namely the minimization of *rep*, *mri* and *mrcl* under the condition that the number of reducers is fixed and set to P. To minimize the corresponding OF, we follow a two-step approach. In the first step, we emphasize on a single-objective, e.g., *mri* or *mrcl*, and we propose the application of agglomerative hierarchical clustering onto JM cells (Sec. 5.2), along with a variety of policies to select the next two cell groups to be merged (Sec. 5.3). In the next step, we explore a larger solution space starting from the solution of the first step with a view to optimizing according to all three weighted criteria (Sec. 5.4). For simplicity, in the remainder of this work, *mri*, $|S|$ and $|T|$ will be measured in histogram buckets and *mrcl* in candidate cells.

5.2 Optimization Through Agglomerative Partitioning

In this section, we describe the first step of our approach, which emphasizes on a single metric. To optimize a single metric, we need to compute the lower and upper bounds. We will first describe the procedure when the main metric, on which we initially focus, is *mri*.

Algorithm 3 nonRectangularSplitting

```
Input: $JM(S \times T),\ P,\ mri$
  $candSet \leftarrow deriveCandidateCells(JM)$
  $groups \leftarrow candSet$
  $numGroups \leftarrow |candSet|$
  while $numGroups > P$ do
    repeat
      $(group1, group2) \leftarrow pickNextGroups(groups)$
      if $(group1, group2) == NULL$ then
        return $false$
      $mergedGroup \leftarrow group1 \cup group2$
    until $|mergedGroup| \leq mri$
    $groups \leftarrow (groups \setminus group1, group2) \cup mergedGroup$
    $numGroups \leftarrow numGroups - 1$
  return $true$
```

Algorithm 2 shows the external procedure, which, similarly to the approach described in Sec. 2.2, tries to minimize only *mri* in a direct manner. The upper bound of *mri* is the total map input, whereas the lower bound is the minimum possible so that all candidate cells can be processed as shown in Algorithm 2 and explained in [21]. For each *mri* value examined, the algorithm aims to derive a grouping of candidate cells that may indirectly target the reduction of additional metrics, such as *rep*. This is done with the help of the procedure in Algorithm 3, which is inspired by agglomerative hierarchical clustering in data mining [13].

A similar binary search-based procedure can be applied regarding $mrcl$. In this case, the upper bound of the binary search is defined as the number of all the candidate cells of the matrix while the lower bound is the upper bound divided by P. Also, the upper and lower bounds for *rep* can be found based on those for *mri* according to the analysis in [16].

nonRectangularSplitting in Algorithm 3 starts regarding each candidate cell in the *JM* as a different partition and it iteratively merges such partitions until P groups remain. The algorithm assumes that the binary search is on *mri*, but it can be modified for *mrcl* in a trivial manner. We assume that candidate cells are more than the number of available reducers, which always holds in non-trivial cases. The algorithm greedily chooses pairs of candidate cells to be merged with the help of *pickNextGroups*. If the merger exceeds the *mri* value under investigation, then the next pair in the ordering is selected to be merged. The same process is repeated until no other pair remains, in which case the algorithm aborts. Otherwise the algorithm continues until exactly P groups are derived.

For each pair of groups, we compute a *distance*. At each merging step, we select the pair with the minimum distance to be merged and subsequently, we update only the affected distances. For the latter, we only update the distances of the pairs affected by the merger. The complexity of our approach depends on a) the dimensions of the *JM*, which is the number of equi-depth histogram buckets, and b) the number of candidate cells. If the maximum

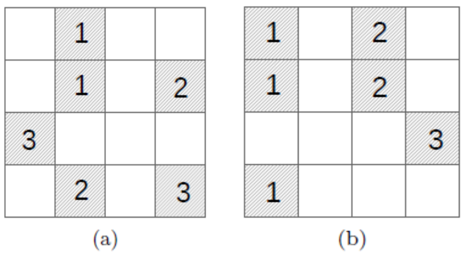


Fig. 8: Example JMs to explain the techniques for distance computation

	Fig. 8a		Fig. 8b	
group	IC	CC	IC	CC
1	3	2	4	3
2	4	2	3	2
3	4	2	2	1
1∪2	5	4	5	5
1∪3	7	4	6	4
2∪3	6	4	5	3

Table 1: The IC and CC metrics for the existing and possible groupings in Figure 8

number of the equi-depth histogram buckets is d and there are c candidate cells, then Algorithm 2 is executed $O(logd)$ times, and the complexity of *non-RectangularSplitting* is $O(c^2logc)$ [13]. Given that in general $c > d$, the overall complexity is $O(c^2logc)$.

A salient feature of this methodology is that we can target different metrics and their combinations through different distance computations, which renders our solution flexible and easily extensible. In other words, through the appropriate selection of the distance function, we can indirectly target additional metrics to the one used in the binary search. Below, we discuss a range of policies to pick groups and evaluate distances that we investigated in this work.

5.3 Policies to pick groups and evaluate distances

Similarly to the re-arrangement methods, the different policies complement each other forming an ensemble (see also Algorithm 1). Consider two 4×4 *JMs*, as shown in Figure 8. Our policies for distance computation are understood with the help of the two low-level metrics per partition $p \in P$, namely input cost (*IC(p)*) and candidate cells (*CC(p)*). In the figures, each cell is annotated with the partition it belongs to. For simplicity, as in all previous examples, let us assume that all buckets have exactly the same number of records, thus counting the number of input buckets is equivalent to counting the number of input records. Table 1 shows the IC and CC values for the partitioning in the figure. For example, in Figure 8b, the IC of the first group is 4 (the 1st, 2nd and 4th bucket of the relation corresponding to the vertical dimension and the 1st bucket of the other relation); if we merge groups 1 and 2, IC increases by 1.

Overall, we explored 7 different ways to group *JM* cells (mentioned as candidate cell clustering solutions in Algorithm 1):

1. *AICS: Added Input Cost - Sum.* we select the pair that, if merged, will have the least increase in *IC* with a view to promoting group mergers that share a high proportion of their *JM* rows and columns so that both *mri* and *rep* stay low. More formally, the distance between two groups $p1$ and $p2$ is given by $AICS(p1 \cup p2) = \frac{IC(p1 \cup p2)}{IC(p1)+IC(p2)}$.
   Consider the example in Figure 8a, where $AICS(1 \cup 2) = \frac{5}{7}$, $AICS(1 \cup 3) = \frac{7}{7}$ and $AICS(2 \cup 3) = \frac{6}{8}$. Then, the groups 1 and 2 will be merged.
2. *AICM: Added Input Cost - Max.* This policy shares the same goal with the previous one, but focuses on the extra input cost for the largest group. The distance formula is given by $AICM(p1 \cup p2) = IC(p1 \cup p2) - max\{IC(p1),\ IC(p2)\}$.
   According to this policy, the groups 1 and 2 in Figure 8a will be merged next as well, since $AICM(1 \cup 2) = 1$, $AICM(1 \cup 3) = 3$ and $AICM(2 \cup 3) = 2$.
3. *ACCM: Added Candidate Cells - Max.* Here, the rationale changes to favor mergers that increase *mrcl* as little as possible. The distance formula is $ACCM(p1 \cup p2) = CC(p1 \cup p2) - max\{CC(p1),\ CC(p2)\}$.
   In the example of Figure 8b, $ACCM(1 \cup 2) = 2$, $ACCM(1 \cup 3) = 1$ and $ACCM(2 \cup 3) = 1$. As such, the algorithm would select either $1 \cup 3$ or $2 \cup 3$ arbitrarily.
4. *EIC: Emptiest Partition - Input Cost.* This policy targets *imb*, which as mentioned earlier combines *mri* and *rep*. In each step, it chooses the partition with the lowest *IC* so far, and it merges with one other partition, which is chosen arbitrarily provided that the threshold is satisfied. The distance function is $EIC(p1 \cup p2) = min\{IC(p1),\ IC(p2)\}$.
   According to Fig. 8a, $IC(1)$ is the lowest value, so the first partition will be merged. Suppose that Algorithm 2 enforces a threshold on *mri* set to 5. Then only $1 \cup 2$ is a valid merger, since $IC(1 \cup 2) = 5$ whereas $IC(1 \cup 3) = 7$.
5. *ECC: Emptiest Partition - Candidate Cells.* This policy shares a similar rationale, but focuses on *CC*. It tries not to allow any partition to have relatively low *CC* , through this distance function: $ECC(p1 \cup p2) = min\{CC(p1),\ CC(p2)\}$.
   In the example of Fig. 8b, the 3rd partition will be merged next because $CC(3) < CC(1), CC(2)$. Suppose that the threshold on *mrcl* is 3. Then $2 \cup 3$ is the only valid choice.
6. *WIC: Weighted dimensions - Input Cost.* This policy, not only favors mergers that share a large proportion of their input, but also favors rectangular partitions, where the long dimension is either the rows or the columns for all the partitions. Trying to shape the distribution of cells across one relation further decreases *rep*. To achieve this, it employs a low-level metric that extends *IC*, called *wIC* given two weights *wR* and *wC* for rows and columns, respectively. For example, if $wR = 2$ and $wC = 1$, in Figure 8a,

$wIC(1) = 5$, because it corresponds to 2 row buckets and 1 column bucket. Then, $WIC(p1 \cup p2) = wIC(p1 \cup p2)$.
Continuing the example, the partitions to be merged are 1 and 2, since $wIC(1 \cup 2) = 8$, $wIC(1 \cup 3) = 11$ and $wIC(2 \cup 3) = 9$.

7. *M: Manhattan distance.* Using the first norm (Manhattan) distance, this policy tries to form clusters of cells that are visually near in the JM. It is based on a similar intuition like the original M-Bucket-I algorithm, that is cells, which are close in the JM, have common inputs. Given two candidate cells c_i and c_j, their manhattan distance *dist* is the number of cells c_i is away from c_j in the horizontal and the vertical dimension of the JM. The exact formula for the distance function is $M(p1 \cup p2) = \frac{1}{|p1||p2|} \sum_{i \in p1} \sum_{j \in p2} dist(c_i, c_j)$, which computes the mean distance between all pairs of candidate cells of the two partitions.
   In Figure 8b, the values are as follows. $M(1 \cup 2) = \frac{19}{6}$, $M(1 \cup 3) = \frac{13}{3}$ and $M(2 \cup 3) = \frac{5}{2}$. Thus, the pair with the smallest distance is 2∪3.

Even though the above policies help us to indirectly affect several metrics and not only *mri* or*mrcl* as binary search only tries, we still need some way to relax the latter's target (*mri* or *mrcl*), to allow for even further improvement of the metrics we are interested in. For this reason, we will next introduce the notion of a post-binary search step, namely range search.

5.4 The complete solution

Algorithm 4 Range Search Solution Exploration

```
Input: $JM(S \times T)$, $P$, $factor$, $k$, $OF$
  $minScore \leftarrow \infty$
  $bestGroups \leftarrow NULL$
  $mriLow \leftarrow$ run Algorithm 2 (or M-Bucket-I)
  $mriRange \leftarrow [mriLow, factor \cdot mriLow]$
  for $mri \leftarrow mriRange$ in steps of size $k$ do
      $groups \leftarrow nonRectangularSplitting(JM,\ P,\ mri)$
      if $OF(groups) < minScore$ then
          $minScore \leftarrow OF(groups)$,  $bestGroups \leftarrow groups$
  return $minScore$, $bestGroups$
```

Using the binary search approach in Algorithm 2 we focus on a single metric, but through the policies in Section 5.3 we affect all metrics in our objective function $OF = \alpha \cdot rep + \beta \cdot mri + \gamma \cdot mrcl$. However, the policies in Section 5.3 run under a potentially strict threshold on the metric optimized through the binary search. Essentially, this leads to limited reductions in the OF.

To ameliorate this problem, instead of calling Algorithm 3 once per each policy in the previous section, we call it multiple times. In each iteration we further relax the constraint in the main metric pursued in Algorithm 2. More specifically, we allow restrictions up to $factor > 1$ times as shown in Algorithm 4 (the algorithm is presented for the case, where the metric in the binary search is mri).

Consider an example where the binary search has returned $mri = 100$ (termed as *mriLow* in Algorithm 4), and the parameters *factor* and k are set to 2 and 10, respectively. This means that we will explore partitionings with mri up to 200 in steps of size 10. I.e., we will explore 11 values of mri : $100, 110, \ldots, 200$. The final partitioning is the one that minimizes the OF and may be produced in any of the $\lceil \frac{(factor-1)mriLow+1}{k} \rceil$ iterations.

5.5 The special case of self joins

Our solutions apply to generic binary joins. However, in the special case of self-join, further considerations can be made. This is due to the fact that, in distance metrics discussed earlier, we should take into account the issue that the row buckets in the JM are the same as the column ones. This implies that a partition requiring the i-th bucket from both inputs will have input cost 1 instead of 2. As such, the way IC is computed and the affected policies in Section 5.3 need to be modified accordingly.

6 Experimental Evaluation

In this section we conduct a thorough evaluation of our proposal, where in total we examined over 7,000 of *JM* partitioning test cases. We first provide a succinct description of the varying dimensions, queries and datasets in our experiments. Initially, we present a few examples comparing actual execution time to better clarify the relationship between OF and running time improvements. The main part of the evaluation contains a detailed presentation of the improvements on the *rep, mri* and *mrcl* metrics. Finally, we present the running time of optimizing the partitioning and a summary of how we propose to apply our approach in practice. The key take-away results are as follows:

1. The techniques are effective in reducing the OF values.
2. Reductions in OF values correspond to tangible benefits in execution time.
3. The form of JM plays a role in the behavior of the techniques.
4. An ensemble approach is preferable indeed; different techniques exhibit the best behavior under different examples.
5. Partitioning and re-arrangement, when combined, do not lead to cumulative benefits.

6.1 Varying dimensions, Queries, Datasets and OFs

The varying dimensions in our experiments include a) the JM re-arrangement method; b) the partitioning method; and c) the number of reducers.

The options for the re-arrangement method are as follows: No re-arrangement (*none*), as in the reference algorithm in [21], *BEA, BEARadius, TSPk* and *TSPkT*, as discussed in Section 4. For BEARadius, and because its impact has not been found to be significant, we present only experiments with the radius set to 3.

The partitioning techniques to be evaluated, include the following. *MBI* (standing for M-Bucket-I described in Section 2 and being the baseline one) and our proposals that produce non-rectangular regions *AICS, AICM, ACCM, EIC, ECC, WIC* and *M*, as discussed in Section 5.3. For *WIC*, we experiment with 3 flavors: *WICa*, where *wR=1, wC=1*; *WICb*, where *wR=1.2, wC=1*; and *WICc*, where *wR=1.5, wC=1.* The total number of partitioning policies is 10. In all experiments, in Algorithm 4, we set *factor* and k to 2 and $\frac{mriLow}{10}$, respectively.

For the number P of reducers, we experimented with the values of 10, 20, 40 and 80, given that most clusters are rather small and have fewer than 50 nodes, as reported in [10,23].

Our policies operate on the initial *JMs* and are agnostic to the actual sizes of the input relations, as their operations apply only onto the *JMs*. In our tests, we have used (i) *band joins* queries over a real dataset to create the initial histograms and *JMs*; and (ii) synthetic *JMs*. All *JM*s have dimensions 100×100, unless otherwise stated (for random queries, we experimented with 200×200 *JMs* as well). Band joins is a specific form of selective theta-joins. In band joins, the theta predicate in $S \bowtie_\theta T$ contains one or more conditions of the form: $S.A - \varepsilon_1 < T.B < S.A + \varepsilon_2$, where A and B are the join attributes and $\varepsilon_1, \varepsilon_2 > 0$.

The real dataset is the Cloud Dataset[6], which is approximately 28.8GB and contains monthly weather reports about clouds from ships and land stations from different parts of the world. The band queries are self-join queries on the *Solar Altitude* and *Longitude* attributes. However, the queries are only used to produce the *JMs*; to evaluate the generic case, the partitioning of the *JMs* was done without exploiting the knowledge that the row and column buckets correspond to the same data. For each distinct configuration setting in terms of re-arrangement, partitioning and number of reducers, we tested queries with number of bands from 1 to 6. The corresponding selectivity varies from 2% to 30% for the band queries on the *Solar Altitude* and from 0.5% to 13% for those on the *Longitude* attribute. For each number of bands, we created 10 random pairs of ε_1 and ε_2; more specifically the ε_1 (resp. ε_2) values are defined by the lower bound of the first (resp. upper bound of the second) out of two randomly picked consecutive histograms buckets. Overall, for each configuration setting, 60 queries are tested. To reduce the number of experiments, we initially test the re-arrangement and the partitioning methods separately (2 attributes $\times$ 5 techniques $\times$ 4 P values $\times$ 60 queries = 2400) and $2 \times 10 \times 4 \times 60 = 4800$ test cases, respectively). We then evaluate their combined efficiency.

[6] available from `http://cdiac.ornl.gov/ftp/ndp026c/`

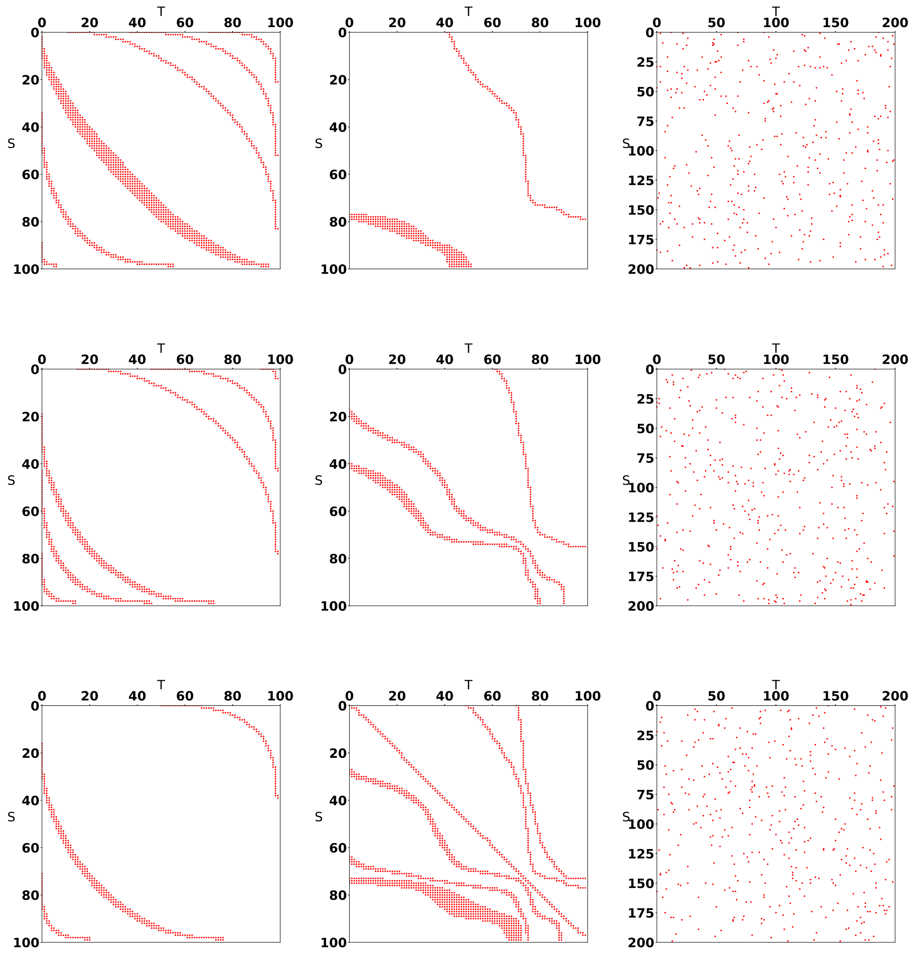


Fig. 9: Nine different example initial JMs for three band-join queries on the solar altitude (left column), three band-join queries on the longitude attribute (middle column), and three random queries (right column).

For the arbitrary queries, we randomly marked 1% of the *JM* cells as candidate cells, i.e., assuming a selectivity of 1%.

In Figure 9, we show example initial 100 × 100 *JMs*; we see that the *JMs* are skewed and differ in their form.

Orthogonally to the configurations, we examine 6 objective functions as shown in Table 2. The weights are chosen in such a way so that M-Bucket-I (MBI) acts as a baseline yielding value 1. In practice, in each experiment, we first run MBI to extract the corresponding *rep, mri* and *mrcl* values, and then we instantiate the OFs. In the first three OFs, each of the bottleneck factors

OF id	α	β	γ	OF id	α	β	γ
OF1	$\frac{1}{x}$	0	0	OF4	$\frac{1}{2x}$	$\frac{0}{y}$	$\frac{1}{2z}$
OF2	0	$\frac{1}{y}$	0	OF5	$\frac{1}{3x}$	$\frac{1}{3y}$	$\frac{1}{3z}$
OF3	0	0	$\frac{1}{z}$	OF6	$\frac{1}{4x}$	$\frac{1}{4y}$	$\frac{1}{2z}$

Table 2: The different objective functions examined. x, y, z are the *rep, mri* and *mrcl* values for M-Bucket-I, respectively, so that M-Bucket-I yields $OF = 1$ always.

	case I	case II	case III
OF ratios			
OF1	1.005	1.011	0.642
OF2	0.994	1.015	0.96
OF3	0.864	0.855	1.032
OF4	0.934	0.933	0.837
OF5	0.954	0.961	0.878
OF6	0.932	0.934	0.916
execution time differences			
small dataset	from 225 to 111 secs	from 92 to 79 secs	from 76 to 76 secs
medium dataset	from 330 to 203 secs	from 246 to 202 secs	from 183 to 168 secs
large dataset	from 658 to 527 secs	from 940 to 902 secs	from 613 to 526 secs

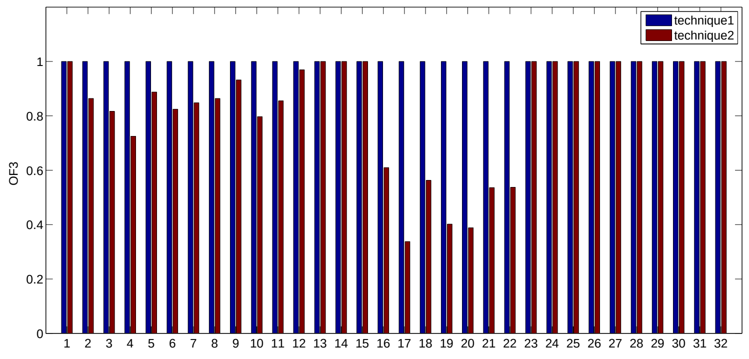
Table 3: Example time differences (in secs) along with the corresponding OF ratios

is regarded in isolation. The last three ones consider the combined effect of these factors.

6.2 Example Time Improvements

In Section 3.1, we have provided relative performance data for different ratios of the three bottleneck metrics, which correspond to the first three OFs. We complement this data with exact execution times for three representative test cases out of the 32 presented in Section 3.1, referred to as case I, II and III in Table 3. This table shows that even small decreases in OF values may correspond to tangible benefits in running times. The running times are common to all OFs, since different OFs just quantify the efficiency of the same partitioning, and therefore the same execution, in a different way. The time differences provided are merely meant to illustrate how the reductions in OF can be correlated to reductions in running time without, at this stage, examining which algorithm yields the lowest OF values.

In a reverse engineering approach, we provide explanations about which OFs would be the most appropriate ones to be selected before execution, should someone wanted to minimize the response time. For the small dataset,



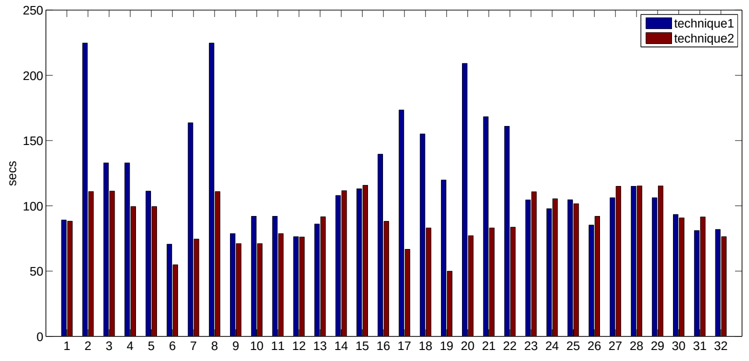
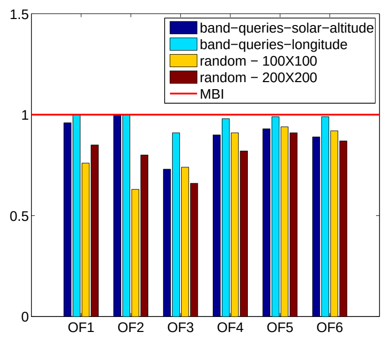


Fig. 10: Top: normalized OF3 values of the 32 pairs in the small dataset in Section 3.1. Bottom: corresponding execution times in secs.

the communication cost and the memory load are not significant, therefore someone could focus on the computational load exclusively, like the rationale of OF3. For the medium and the larger datasets, this does not hold and the objective function should target all three metrics, such as OF4-6. Figure 10 shows the relationship between the OF3 values and execution times for all 32 pairs in the small dataset. As previously, the exact technique that yielded these results does not matter; therefore, in each pair, they are referred to as *technique1* and *technique2*. Fine-tuning the OF to accurately reflect the execution time of these queries on our infrastructure at BSC is out of the scope of this research; further discussion about the appropriate choice of an OF for a specific query is deferred to the end of this section. However, the main observation that is important to our work is that by reducing the appropriately set OFs, we manage to decrease query times as well.



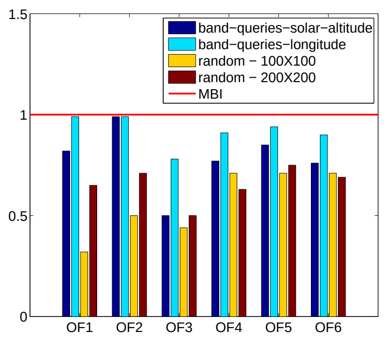


Fig. 11: Average (left) and maximum (right) improvements on the values of OFs when P=80.

6.3 Improvements on OF values due to re-arrangements

We aim to answer three questions: (a) How large are the decreases in the OF value? Remember that there can never be an increase; eventually, the best performing technique is chosen only if it is beneficial. (b) How frequently do our techniques lead to OF decrease? (c) Do we really need many techniques?

The first set of experiments investigates the efficiency of the matrix re-arrangement policies in Section 4. Figure 11 shows the reduction in OFs for $P = 80$. Regarding band queries on solar altitude, the two main observations are that the maximum improvements in an isolated test case can reach 50%. These correspond to OF3, which targets *mrcl* exclusively. The smaller improvements are for OF2; this is expected, since the latter is directly addressed by M-Bucket-I. The maximum average improvement in all cases grouped by OF is *at least* 27%, given that the lowest average OF value is 0.73. Similarly, for the queries on the longitude attribute, the highest improvements are observed for OF3, but they are of lower magnitude, since, in an isolated case the can reach 22% for $P = 80$. For random queries, higher improvements are observed. E.g., *rep* values can drop by up to 68% in the figure, whereas in the results in the Appendix, which cover also 10-40 reducers, the *rep* decrease is up to 74%.

Regarding the frequency of improvements, the band queries on the two attributes are improved 54% and 50% of the times, respectively. Random queries are improved 76% and 99% of the times for *JMs* of size 100×100 and 200×200, respectively. More detailed results are in the Appendix A.

Up to this point, we focused on improvements due to any technique. We now shift our attention to explain our choice to adopt an ensemble methodology. Table 4 shows the percentage of cases in the band queries on solar altitude, where each of the re-arrangement techniques is the best, considering only the cases, where there is an improvement on original *MBI*. We can

	OF1	OF2	OF3	OF4	OF5	OF6
BEA	18.70	17.09	29.85	19.71	22.22	24.09
BEARadius	18.70	18.80	38.06	26.28	21.43	23.36
TSPk	53.66	58.12	30.60	37.96	42.06	35.77
TSPkT	8.94	6.84	27.61	16.06	14.29	16.79
within 5% of the best						
BEA 5%	56.10	60.68	40.30	54.01	56.35	50.36
BEARadius 5%	51.22	53.85	44.78	53.28	46.83	51.09
TSPk 5%	78.86	77.78	38.81	61.31	68.25	62.04
TSPkT 5%	32.52	35.04	34.33	38.69	36.51	37.96
TSPk/TSPKT 5%	85.37	86.32	58.21	74.45	80.95	75.91

Table 4: Percentage of cases where each of the proposed re-arrangement techniques is the best in band queries on solar altitude.

see that there is no clear winner, which means that all techniques contribute to the performance improvements. Nevertheless, *TSPk* leads to improvements more frequently. If we relax our optimality criteria, and we tolerate up to 5% deviation from the best performing policy, then, as shown in the lower part of the table, *TSPk* dominates in approximately 2/3 of the cases. The last row in this table shows that in 76.9% at least one of the two TSP-based solutions is up to 5% far from the best, i.e., they are better than the BEA-based ones. In random queries with a 100×100 *JM*, the behavior is similar: *TSPk* is again the dominating one, whereas, at least one of the two TSP-based solutions yields the highest improvements in 84.6% of the cases. However, in the experiments with 200×200 *JM*, *BEA* is better in 80% of the cases. Finally, *BEA* and TSP-based solutions complement each other in the band query tests on longitude. Therefore, both *BEA* and TSP-based solutions need to be checked in the generic case. Another generic remark is that the behavior of the re-arranging policies differs with the form of the *JMs*.

6.4 Improvements on OF values due to partitionings

Our partitionings are shown to be more efficient than matrix-re-arrangement for band queries. Figure 12(top), which refers to the band query on the solar altitude attribute, shows how the average and lowest OF values differ between re-arrangement and partitioning policies for P=10 and P=80. According to the figure, due to the partitionings, the maximum reduction in *rep* (OF1), which means lower communication cost, is 40%, while the maximum reduction in *mrcl* (OF3) stays at 50%. Figure 13 shows an example, where *rep* is decreased by 40% compared to MBI. On the left part, each rectangular region corresponds to a different reducer. On the right part, the bold points correspond to a single reducer, to provide an insight in the type of the new partitioning. Finally, the frequency of improvements increased to 83% (from 54%). The improvements are more significant for the queries on the longitude attribute, as shown in the bottom part of the figure; *rep* is decreased by up to 45% (OF value 0.55)

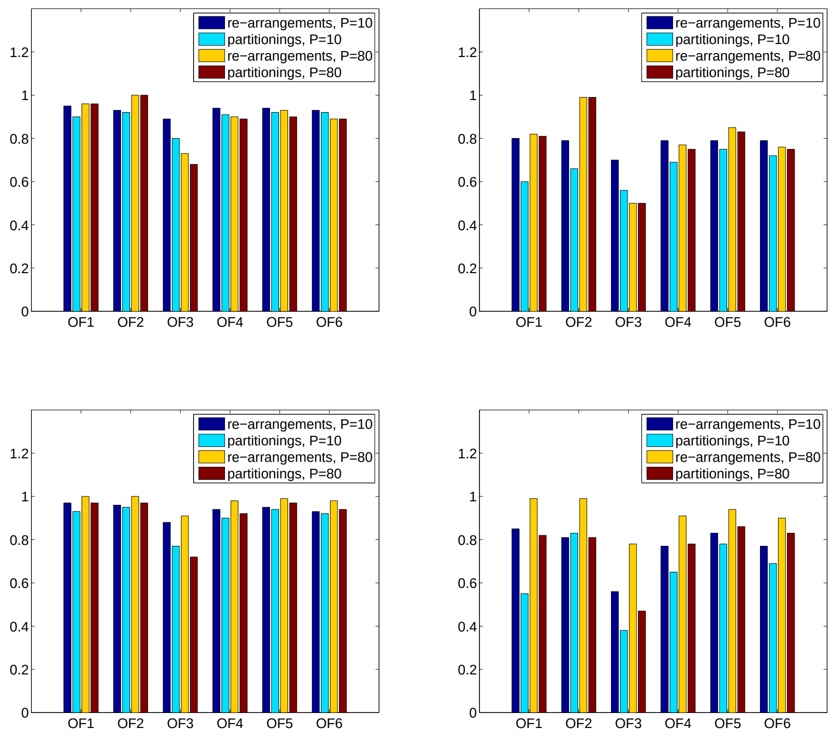


Fig. 12: Average (left) and minimum (right) values of OFs when P=10 and P=80 for band queries (top: solar altitude, bottom: longitude).

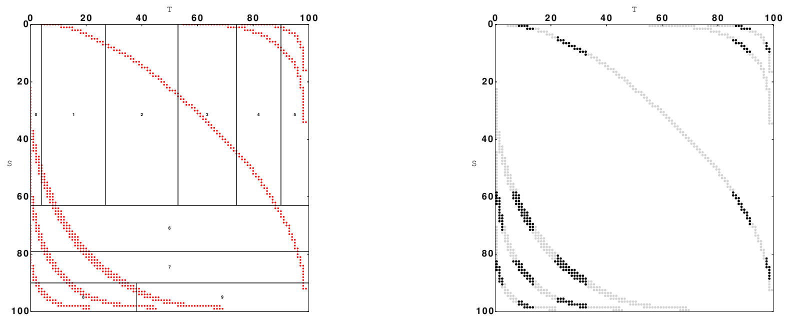


Fig. 13: Example improvement of the initial partitioning due to MBI (left) with the help of *WICa* (right); in the latter figure, the bold points belong to a single partition, as an example of the new partitioning.

compared to MBI. They are also more frequent occurring in 88% of the test cases.

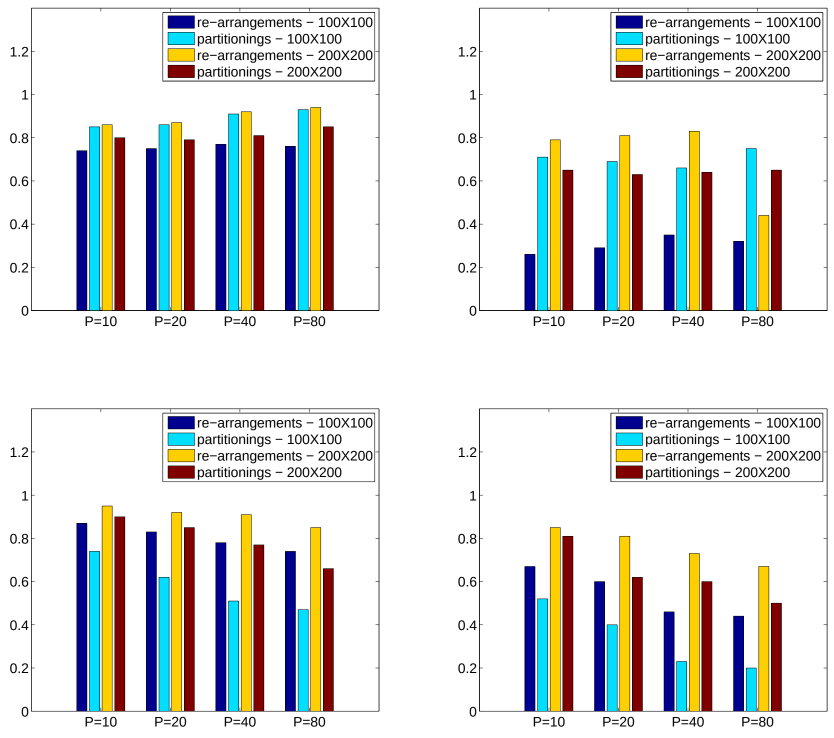


Fig. 14: Average (left) and minimum (right) values of OF1 (top) and OF3 (bottom) for random queries.

For random queries, the results are mixed. The partitioning approaches are more efficient in general, but not always. E.g., in Figure 14, we show comparisons regarding OF1 and OF3, and re-arrangements behave better regarding OF1 when the *JM* dimensions are 100×100. Another interesting observation is that the highest improvements in isolated cases are even more significant for *mrcl*, which is dropped up to the 20% of its original value.

Regarding the rationale to follow an ensemble approach, as shown in the upper part of the Table 5, again, there is no winner among the different proposed techniques, and each of them may be the most efficient in certain cases thus calling for an ensemble method. Column values may sum to higher than 100 because of ties. However, *AICM, AICS* and *WIC* are the dominating ones. This is reflected by the numbers in the third row from the bottom, which shows the proportion of the cases where any of the three aforementioned techniques leads to the best OF results; this proportions is over 97% overall. If we allow up to 5% degradation, the proportion increases to over 99% (bottom row). In general, the best performing stand-alone technique is *AICS*, which is either the optimal or up to 5% worse than the optimal in 68% of the cases (penultimate row). The numbers are for the band queries on the solar altitude attribute, but

	OF1	OF2	OF3	OF4	OF5	OF6
AICM	3.45	18.45	13.97	21.89	33.71	29.95
AICS	72.84	19.42	11.35	54.08	56.00	46.08
ACCM	0.43	6.80	96.94	3.86	4.57	3.69
EIC	0.43	6.80	99.56	3.86	4.57	3.69
ECC	0.43	6.80	100.00	3.86	4.57	3.69
M	0.43	3.88	10.92	5.15	8.57	5.99
WICa	23.71	65.05	100	26.18	13.71	19.82
WICb	0.86	7.77	100	6.01	6.86	10.60
WICc	0.86	7.77	100	6.44	6.86	12.44
AICS/AICM/WICa	100	98.06	100	95.28	93.71	88.02
within 5% of the best						
AICS 5%	86.21	58.25	12.23	90.13	84.57	78.80
AICS/AICM/WICa 5%	100	100	100	100.00	98.86	99.08

Table 5: Percentage of cases where each of the proposed partitioning techniques is the best (band queries)

they hold for the other *JM* types, i.e., those for the queries on longitude and the random queries, with negligible differences. The key implication of this observation is that a simplified ensemble method could contain fewer policies, namely the *AICM, AICS* and *WIC* ones only.

6.5 Improvements due to both re-arrangements and partitionings

A question may arise as whether combining the re-arrangement and partitioning techniques yields higher improvements. For band queries, on average, the combination of techniques yields improvements very close to the improvements achieved by the proposed partitioners only; the benefits are less than 2%. For random queries, there are no benefits. The lesson learnt is that, although it is beneficial to use multiple techniques for re-arrangement and multiple techniques for partitioning, combining the two types can be skipped. In other words, in Algorithm 1, the two loops can be placed in sequence rather than in a nested manner, which incurs lower time overhead without significant sacrifice in the partitioning quality. Overall, efficiently combining re-arrangement with partitioning is an interesting direction for future work.

6.6 Running time of our approach

If someone would test all 5 re-arrangements and 10 partitioning combinations, the time overhead of preparing the final partitioning is non-negligible. Here, we provide exact numbers, when all techniques run on a machine with Intel Core i5-4690 CPU at 3.5GHz and 8 GBs RAM. Table 6 shows the average running time for the partitioning techniques per number of reducers. Note that based on our experimental evidence, a simplified ensemble method would consider only *TSPk, BEA, AICS, AICM* and *WIC*. A single application of them incurs

technique	P=10	P=20	P=40	P=80
MBI	167	162	158	99
AICM	1656	695	672	607
AICS	1526	624	593	521
ACCM	3650	2513	2839	3183
EIC	1862	1777	1857	1836
ECC	2140	1845	2225	2999
M	629	603	595	559
WIC	1196	1065	1091	1285
BEA	155			
BEARadius	197			
TSPk	4796			
TSPkT	9455			

Table 6: Average running time of partitioning techniques (in msecs)

an overhead of approximately 7 secs for 20 reducers on our machine. Moreover, their execution can be parallelized in a straight-forward manner.

6.7 Putting everything together

Here, we summarize our end-to-end proposal for join execution. This discussion is an elaboration of Figure 4 and capitalizes on our evaluation results, according to which a simplified ensemble method, where only a subset of the re-arrangement and partitioning techniques need to be applied, is sufficient. We split parallel theta-join execution in three phases and this proposal targets the second phase:

1. Preparatory phase:
   (a) Construct the *JM*, e.g., adopting the approach in [21].
   (b) Run M-Bucket-I to evaluate the *rep, mri*, and *mrcl* metrics.
   (c) Choose an objective function using the results of the M-Bucket-I runs to normalize the weights as shown in Section 6.1.
2. Partitioning optimization phase (our contribution):
   (a) Run Algorithm 1 either in full or only for *TSPk, BEA, AICS, AICM* and *WIC*.
   (b) Assess improvements compared to M-Bucket-I.
3. Actual join execution phase.

As can be inferred by the above summary, in order to apply our proposal in practice in a beneficial manner, the OF needs to be selected judiciously. Since the criteria values are not at the same scale, we showed a way to normalize them. Still, the relative weights need to be chosen as well. Here, we propose an empirical solution to this problem and we leave the in-depth investigation as future work. In line with the discussion of Table 3, the empirical solution is to assign the *rep, mri*, and *mrcl* weights in proportion to the estimated relative impact of the shuffling cost, memory stress and in-memory evaluation of the local theta-joins on the total execution time, respectively. Note that such an impact has been shown to be dependent on several query, dataset and infrastructure characteristics.

7 Related Work

Binary theta joins in a massively parallel setting is an area that has attracted a lot of interest. As presented in detail in Section 2, the reference algorithms are 1-Bucket-Theta for the generic case and M-Bucket-I/O for selective theta joins [21]. Our solutions significantly extend these algorithms and provide both more efficient solutions and solutions that can consider more optimization objectives, namely replication factor, reducer input and reducer computational load.

The work in [12] presents an adaptive solution for theta joins, but the solution considers only the reducer input. The key feature of this work is that it assumes a streaming environment and may revise the partitioning on the fly. Theta joins are also dealt with in [14], which focuses on algorithms for efficient predicate evaluation regardless of whether the theta join evaluation takes place in a parallel manner. The proposed solutions are based on pre-sorting and data structures such as permutation arrays and bitmaps. The issues of streaming data and sophisticated predicate evaluation are orthogonal to our partitioning approach.

Other efforts to extend the work in [21] have been made in the direction of efficient execution of multi-way theta-joins [4,30,7,28,29], or specific forms of joins, such as star-joins [6]. Exploring more flexible partitioning for multi-way theta joins is not addressed though, and this topic remains an interesting open issue for future research. [27] explores partitioning targeting both *mri* and *mrcl*, but still considers only rectangular matrix partitionings and focuses on a specific form of JMs termed as monotonic, which correspond to 1 band theta joins; i.e. it is less generic than our work.

Theta-joins are also related to similarity joins, e.g. [24]. However, the main effort in similarity joins in MapReduce is to prune non-relevant pairs as soon as possible, which corresponds to eliminating *JM* candidate cells rather than allocating work to reducers in a multi-objective manner after having produced the candidate cells. Finally, [8] examines a specific form of self-joins for data deduplication and considers *JM* cells of different sizes, whereas, in our case, *JM* cells are of the same size due to the equi-depth histograms employed.

Regarding the theoretical analysis of MapReduce programs, most of the proposals for programs in a massively parallel setting to date tend to be developed on a best-effort basis, without systematically analyzing the inherent trade-offs. Two recent remedies to that have been proposed in [2,25]. [25] introduces the notion of *minimal* MapReduce algorithms, which are algorithms accompanied by guarantees (up to a small constant) regarding several aspects, such as memory consumption and communication cost. The MapReduce rounds may be bounded but they can be more than one. The work in [2] is complementary and presents a way to compute the lower bounds on communication cost as a function of the maximum input a reducer is allowed to receive for specific problems. This allows to define the trade-off between the load on the reducer side and the replication rate. Further, the work in [2] examines whether known algorithms for those problems can match the lower bounds, provided that they consist of a single MapReduce round. The work in [16] builds upon the work in [2] to prove the optimality of 1-Bucket-Theta, while [1] examines the theoretical properties of self-joins and proposes a partitioning considering *rep* and *mri* for this specific case.

8 Conclusions and Further Work

In this work, we introduce new partitioning techniques, operating as an ensemble, in order to boost the performance of theta-joins executed in massively parallel environments, such as Spark. To this end, we propose re-arranging the join matrix, and splitting it to non-rectangular partitions. A key feature of our work is that the exact definition of performance is flexible, in that it is capable of accounting for any weighted combination of the main bottleneck factors, namely communication cost and computational and memory load on the reducers. In addition, the impact of our partitioning on the bottleneck factors can be accurately computed before execution. The evaluation results show that significant improvements, up to 5 times, can be achieved.

Interesting directions for future work include the investigation of more efficient combination of matrix re-arrangement and partitioning, and multi-way theta-joins. Another line of research already mentioned is to automatically adjust the objective function weights so that the running time is minimized; as shown in Section 3.1, under different queries, datasets and cluster settings, the relative importance of each of the factors changes, and it is challenging to estimate the corresponding weights a-priori so that they accurately reflect the execution time.

Acknowledgements We would like to thank Jordi Torres, Rubèn Tous and Carlos Tripiana from the Barcelona Supercomputing Center for their help in running the Spark experiments.

References

1. Foto Afrati and Jeffrey Ullman. Matching bounds for the all-pairs mapreduce problem. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 3–4. ACM, 2013.
2. Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.
3. Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.
4. Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

5. HM Chan and DA Milner. Direct clustering algorithm for group formation in cellular manufacture. *Journal of Manufacturing Systems*, 1(1):65–75, 1982.
6. Shih-Ying Chen, Tsui-Ping Chang, and Zhi-Hong Chang. An efficient theta-join query processing algorithm on mapreduce framework. In *Computer, Consumer and Control (IS3C), 2012 International Symposium on*, pages 686–689. IEEE, 2012.
7. Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78, 2015.
8. Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.
9. Sharlee Climer and Weixiong Zhang. Rearrangement clustering: Pitfalls, remedies, and applications. *The Journal of Machine Learning Research*, 7:919–943, 2006.
10. Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.
11. Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, pages 1–26, 2013.
12. Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.
13. Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
14. Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Lightning fast and space efficient inequality joins. *PVLDB*, 8(13):2074–2085, 2015.
15. James R King. Machine-component grouping in production flow analysis: an approach using a rank order clustering algorithm. *International Journal of Production Research*, 18(2):213–232, 1980.
16. Ioannis Koumarelas, Athanasios Naskos, and Anastasios Gounaris. Binary theta-joins using mapreduce: Efficiency analysis and improvements. In *Proceedings of the International Workshop on Algorithms for MapReduce and Beyond (BMR) (in conjunction with EDBT/ICDT'2014)*, Athens, Greece, 2014.
17. Jan K Lenstra and AHG Rinnooy Kan. Some simple applications of the travelling salesman problem. *Operational Research Quarterly*, pages 717–733, 1975.
18. JK Lenstra. Technical noteclustering a data array and the traveling-salesman problem. *Operations Research*, 22(2):413–414, 1974.
19. Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31, 2014.
20. William T McCormick, Paul J Schweitzer, and Thomas W White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20(5):993–1009, 1972.
21. Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD Conference*, pages 949–960, 2011.
22. Alper Okcan and Mirek Riedewald. Anti-combining for mapreduce. In *SIGMOD Conference*, pages 839–850, 2014.
23. Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence. *PVLDB*, 6(10):853–864, 2013.
24. Akash Das Sarma, Yeye He, and Surajit Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.
25. Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *SIGMOD Conference*, pages 529–540, 2013.
26. Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguade, Jesus Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *IEEE BigData*, 2015.
27. Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. Load balancing and skew resilience for parallel joins. In *Proc. of ICDE*, 2016.
28. Ke Yan and Hong Zhu. Two MRJs for multi-way theta-join in mapreduce. In *Internet and Distributed Computing Systems*, pages 321–332. Springer, 2013.

OF	coverage	avg	best	worst	imb
$P = 10$					
OF1	85.00%	0.95	0.80	1.00	-0.71%
OF2	83.33%	0.93	0.79	1.00	-1.57%
OF3	88.33%	0.89	0.70	0.99	-1.27%
OF4	91.67%	0.94	0.79	1.00	-1.09%
OF5	88.33%	0.94	0.79	1.00	-1.43%
OF6	93.33%	0.93	0.79	1.00	-1.29%
$P = 20$					
OF1	70.00%	0.96	0.87	1.00	+0.35%
OF2	65.00%	0.96	0.88	1.00	-0.84%
OF3	71.67%	0.91	0.67	0.98	-0.28%
OF4	71.67%	0.96	0.85	1.00	-0.17%
OF5	66.67%	0.96	0.88	1.00	-0.57%
OF6	73.33%	0.96	0.86	1.00	-0.41%
$P = 40$					
OF1	26.67%	0.97	0.84	1.00	+2.94%
OF2	28.33%	0.97	0.83	1.00	-0.79%
OF3	35.00%	0.84	0.50	0.98	+1.58%
OF4	36.67%	0.94	0.77	1.00	+2.14%
OF5	30.00%	0.95	0.85	1.00	-0.03%
OF6	35.00%	0.94	0.76	1.00	+0.99%
$P = 80$					
OF1	23.33%	0.96	0.82	1.00	+3.15%
OF2	18.33%	1.00	0.99	1.00	-1.46%
OF3	28.33%	0.73	0.50	0.95	+0.46%
OF4	28.33%	0.90	0.77	1.00	+0.75%
OF5	25.00%	0.93	0.85	1.00	-1.14%
OF6	26.67%	0.89	0.76	1.00	+0.43%

Table 7: Summary improvements due to the re-arrangement policies grouped by the number of reducers (for band queries)

29. Changchun Zhang, Jing Li, and Lei Wu. Optimizing theta-joins in a mapreduce environment. *International Journal of Database Theory & Application*, 6(4), 2013.
30. Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.

A Additional Evaluation Results

Tables 7 and 8 refer to the experiments in Section 6.3 for the band queries on solar altitude. Table 8 presents the same results as Table 7 but groups the experiments differently to show the impact of selectivity (in terms of number of bands). Although the behavior differs according to the number of bands, the impact of selectivity is considered to be small. The second column (coverage) shows the percentage of the cases, where an improvement on M-Bucket-I is achieved by any technique, i.e., it answers the question *"How frequently matrix re-arrangement leads to improvements?"*, whereas the other columns answer the question *"How high are the improvements when they happen?"*. Further observations that can be drawn are: (i) The higher the number of reducers, the less frequently matrix re-arrangement yields improvements. (ii) The benefits on OF values due to the re-arrangement techniques may come at the expense of a small degradation of imbalance, as shown in the last column, but in general *imb* is not affected much. (iii) There are several cases where the improvement is very small or negligible. Table 9 shows the corresponding details for the band queries on longitude, where the best improvements on *mrcl* are up to 44%. Table 10 shows the impact of the re-arrangement techniques on the OFs for random queries with 100×100 *JM*. The main observation is that, compared to Table 7, both the coverage and the improvements are higher; e.g., we have observed reductions in *rep* by 74% (i.e., nearly 4 times less) and in *mrcl* by 56%. For random queries with 200×200 *JMs*, the improvements are of lower magnitude but the coverage is 88% (no detailed results are presented).

OF	coverage	avg	best	worst	imb
$bands = 1$					
OF1	62.50%	0.96	0.82	1.00	-0.68%
OF2	65.00%	0.96	0.83	1.00	-2.42%
OF3	45.00%	0.75	0.50	0.97	-3.07%
OF4	62.50%	0.92	0.77	1.00	-1.69%
OF5	67.50%	0.94	0.85	1.00	-1.68%
OF6	67.50%	0.92	0.76	1.00	-1.68%
$bands = 2$					
OF1	50.00%	0.97	0.93	1.00	+1.09%
OF2	47.50%	0.97	0.87	1.00	-1.39%
OF3	62.50%	0.86	0.50	0.98	+0.91%
OF4	62.50%	0.94	0.77	1.00	+1.07%
OF5	55.00%	0.95	0.85	1.00	-0.93%
OF6	65.00%	0.94	0.76	1.00	+0.65%
$bands = 3$					
OF1	47.50%	0.97	0.89	1.00	+1.17%
OF2	35.00%	0.96	0.88	1.00	-1.25%
OF3	50.00%	0.87	0.60	0.99	-0.74%
OF4	57.50%	0.95	0.81	1.00	+0.04%
OF5	47.50%	0.95	0.87	1.00	-1.41%
OF6	55.00%	0.95	0.80	1.00	-0.82%
$bands = 4$					
OF1	40.00%	0.96	0.90	0.99	+0.42%
OF2	37.50%	0.96	0.80	1.00	-0.39%
OF3	57.50%	0.90	0.77	0.98	+0.16%
OF4	47.50%	0.95	0.84	1.00	+0.07%
OF5	40.00%	0.95	0.83	1.00	-0.55%
OF6	42.50%	0.94	0.81	1.00	-0.26%
$bands = 5$					
OF1	57.50%	0.95	0.80	1.00	+0.21%
OF2	55.00%	0.93	0.79	1.00	-0.51%
OF3	57.50%	0.89	0.70	0.98	-0.24%
OF4	57.50%	0.93	0.79	1.00	-0.26%
OF5	55.00%	0.93	0.79	1.00	-0.35%
OF6	57.50%	0.93	0.79	1.00	-0.42%
$bands = 6$					
OF1	50.00%	0.94	0.82	1.00	-0.47%
OF2	52.50%	0.93	0.79	1.00	-1.40%
OF3	62.50%	0.90	0.73	0.99	-0.60%
OF4	55.00%	0.94	0.81	1.00	-0.61%
OF5	50.00%	0.94	0.82	1.00	-1.02%
OF6	55.00%	0.93	0.81	1.00	-0.88%

Table 8: Summary improvements due to the re-arrangement policies grouped by the number of bands (for band queries)

OF	coverage	avg	best	worst	imb
$P = 10$					
OF1	70.00%	0.97	0.85	1.00	-0.50%
OF2	66.67%	0.96	0.81	1.00	-2.16%
OF3	78.33%	0.88	0.56	1.00	-1.78%
OF4	75.00%	0.94	0.77	1.00	-1.81%
OF5	75.00%	0.95	0.83	1.00	-2.02%
OF6	75.00%	0.93	0.77	1.00	-1.96%
$P = 20$					
OF1	46.67%	0.98	0.90	1.00	-0.65%
OF2	48.33%	0.97	0.89	1.00	-2.99%
OF3	56.67%	0.89	0.67	0.99	-2.38%
OF4	61.67%	0.96	0.84	1.00	-2.08%
OF5	53.33%	0.97	0.86	1.00	-2.79%
OF6	61.67%	0.96	0.82	1.00	-2.42%
$P = 40$					
OF1	21.67%	0.99	0.96	1.00	-1.37%
OF2	30.00%	0.98	0.87	1.00	-1.98%
OF3	58.33%	0.88	0.69	0.98	-0.71%
OF4	58.33%	0.96	0.88	1.00	-0.86%
OF5	50.00%	0.97	0.90	1.00	-1.35%
OF6	56.67%	0.96	0.87	1.00	-1.19%
$P = 80$					
OF1	13.33%	1.00	0.99	1.00	+0.34%
OF2	18.33%	1.00	0.99	1.00	-1.12%
OF3	35.00%	0.91	0.78	0.97	-1.83%
OF4	35.00%	0.98	0.91	1.00	-1.72%
OF5	26.67%	0.99	0.94	1.00	-2.08%
OF6	36.67%	0.98	0.90	1.00	-2.13%

Table 9: Summary improvements due to the re-arrangement policies grouped by the number of reducers (for band queries)

OF	coverage	avg	best	worst	imb
$P = 10$					
OF1	100.00%	0.74	0.26	0.99	+0.69%
OF2	100.00%	0.75	0.38	0.92	+0.41%
OF3	66.00%	0.87	0.67	0.96	+0.06%
OF4	96.00%	0.89	0.66	1.00	+0.53%
OF5	100.00%	0.87	0.63	1.00	+0.60%
OF6	96.00%	0.90	0.68	1.00	+0.49%
$P = 20$					
OF1	100.00%	0.75	0.29	0.99	+0.36%
OF2	100.00%	0.77	0.44	0.86	-0.06%
OF3	53.00%	0.83	0.60	0.93	-0.04%
OF4	86.00%	0.91	0.69	1.00	+0.29%
OF5	95.00%	0.90	0.68	1.00	+0.16%
OF6	81.00%	0.92	0.73	1.00	+0.20%
$P = 40$					
OF1	100.00%	0.77	0.35	0.99	-0.91%
OF2	49.00%	0.74	0.60	0.75	-1.29%
OF3	51.00%	0.78	0.46	0.89	+0.60%
OF4	76.00%	0.90	0.65	1.00	+0.21%
OF5	82.00%	0.91	0.66	1.00	+0.09%
OF6	74.00%	0.90	0.66	1.00	+0.21%
$P = 80$					
OF1	100.00%	0.76	0.32	0.99	-2.73%
OF2	2.00%	0.63	0.50	0.67	-13.25%
OF3	38.00%	0.74	0.44	0.88	-1.22%
OF4	60.00%	0.91	0.71	1.00	-2.08%
OF5	61.00%	0.94	0.71	1.00	-2.01%
OF6	56.00%	0.92	0.71	1.00	-1.70%

Table 10: Summary improvements due to the re-arrangement policies grouped by the number of reducers (for random queries)